Inf1-OP
Collections

Volker Seeker, adapting earlier version by Perdita Stevens and Ewan Klein

School of Informatics

February 26, 2018
Rigidity of arrays

- Length of array is fixed at creation time.
- Can’t be expanded.
- Can’t be shrunk.
- Arrays are part of Java language — uses special syntax.
- E.g., `myArray[i]` for accessing the ith element.
ArrayList:

- Can grow and shrink as needed;
- Provides methods for inserting and removing elements.

Declaration

```java
ArrayList<String> cheers = new ArrayList<String>();
```

- This is an array list of strings; counterpart to `String[]`.
- Angle brackets indicate that `String` is a type parameter.
- Can replace `String` with e.g. `HotelRoom` to get different array list type.
- In general: use `ArrayList<E>` to collect objects of type `E`; but `E` cannot be a primitive type.
ArrayList: Methods

- A newly constructed ArrayList has size 0.
- ArrayList has various methods, which allow us to:
  - keep on adding new elements;
  - remove elements.
- The size changes after each addition / removal.
ArrayList: Adding

Adding Elements

ArrayList<String> cheers = new ArrayList<String>();
cheers.add("hip");
cheers.add("hip");
cheers.add("hooray");
int n = cheers.size(); // n gets value 3

▶ add() appends each element to the end of the list.

Printing an ArrayList

System.out.println(cheers);

Output

[hip, hip, hooray]
ArrayList: More methods

Index of first occurrence

```java
int ind = cheers.indexOf("hip"); // ind gets value 0
```

Adding element at an index

```java
cheers.add(1, "hop"); // 2nd "hip" gets shunted along
```

Elements of cheers: ["hip", "hop", "hip", "hooray"]
ArrayList: More methods

contains()

```java
boolean isHip = cheers.contains("hip"); // isHip is true
```

remove()

```java
cheers.remove("hip"); // removes first occurrence of "hip"
```

Elements of cheers: "hop", "hip", "hooray"

get(int index)

```java
cheers.get(0); // get the first element
// returns "hop"
```
ArrayList and Loops

Looping over ArrayList:

Standard for loop

```java
for (int i = 0; i < cheers.size(); i++) {
    System.out.println(cheers.get(i));
}
```

Enhanced for again

```java
for (String s : cheers) {
    System.out.println(s);
}
```
ArrayList and Loops

Looping over ArrayList:

**Standard for loop**

```java
for (int i = 0; i < cheers.size(); i++) {
    System.out.println(cheers.get(i));
}
```

**Enhanced for again**

```java
for (String s : cheers) {
    System.out.println(s);
}
```
ArrayList and Loops

Enhanced for again

```java
for (String s : cheers) {
    System.out.print(s + "\thas index: ");
    System.out.println(cheers.indexOf(s));
}
```

Output

- hop has index: 0
- hip has index: 1
- hooray has index: 2
Wrapper Classes

The type variable E in a generic type like ArrayList<E> must resolve to a reference type.

So ArrayList<int> will not compile.

All the primitive types can be turned into objects by using wrapper classes:

```
<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
</tbody>
</table>
```

NB Wrapper class names are always capitalized, always complete words.
Auto-boxing

- Conversion between primitive types and corresponding wrapper classes is automatic.
- Process of conversion is called auto-boxing

**Auto-box example**

```java
Double batteryCharge = 2.75;
double x = batteryCharge;
```

**Auto-box example**

```java
ArrayList<Double> data = new ArrayList<Double>();
data.add(29.95);
double x = data.get(0);
```
Import

Importing:

▶ To get full access to Java API, we need to import classes.
▶ Not necessary if class is in same folder, or part of java.lang (e.g., Math library).
▶ To use ArrayList, add the appropriate import statement at top of your file:

Import example

```java
import java.util.ArrayList;
```
Importing:

- To get full access to Java API, we need to import classes.
- Not necessary if class is in same folder, or part of java.lang (e.g., Math library).
- To use ArrayList, add the appropriate `import` statement at top of your file:

Import example

```java
import java.util.ArrayList;
```

Import example — Wrong!

```java
import java.util.ArrayList<String>; // Don’t use parameter
```
Java API

Look at sample Javadoc web page.
http://docs.oracle.com/javase/8/docs/api/
Java™ Platform, Standard Edition 6
API Specification

This document is the API specification for version 6 of the Java™ Platform, Standard Edition.

See:

<table>
<thead>
<tr>
<th>Packages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.applet</td>
<td>Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.</td>
</tr>
<tr>
<td>java.awt</td>
<td>Contains all of the classes for creating user interfaces and for painting graphics and images.</td>
</tr>
<tr>
<td>java.awt.color</td>
<td>Provides classes for color spaces.</td>
</tr>
<tr>
<td>java.awt.datatransfer</td>
<td>Provides interfaces and classes for transferring data between and within applications.</td>
</tr>
<tr>
<td>java.awt.dnd</td>
<td>Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.</td>
</tr>
<tr>
<td>java.awt.event</td>
<td>Provides interfaces and classes for dealing with different types of events fired by AWT components.</td>
</tr>
<tr>
<td>java.awt.font</td>
<td>Provides classes and interface relating to fonts.</td>
</tr>
</tbody>
</table>
Class `ArrayList<E>`

```java
java.lang.Object
  | java.util.AbstractCollection<E>
  |   java.util.AbstractList<E>
  |     java.util.ArrayList<E>
```

All Implemented Interfaces:
- `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess`

Direct Known Subclasses:
- `AttributeList`, `RoleList`, `RoleUnresolvedList`

```java
public class ArrayList<E>
  extends AbstractList<E>
  implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding `n` elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Each `ArrayList` instance has a `capacity`. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an `ArrayList`, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

An application can increase the capacity of an `ArrayList` instance before adding a large number of elements using the `ensureCapacity` operation. This may reduce the amount of incremental reallocation.
Associative Arrays

Associative array:

- Associates a collection of unique keys with values.
- Ordinary arrays: keys can only be integers.
- Associative arrays allow keys of many types, most notably strings.
- Examples:
  1. Given a person’s name, look up a telephone number.
  2. Given an internet domain, look up its IP address.
  3. Given a geo-location, look up its GPS coordinates.
  4. Given a word, look up its frequency in a text.

- Relationship between key and value: mapping.

Java: associative arrays are implemented by type HashMap.
Map People to their Matric Nos.

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>s0189034</td>
</tr>
<tr>
<td>Michael</td>
<td>s0289125</td>
</tr>
<tr>
<td>Helen</td>
<td>s0378435</td>
</tr>
<tr>
<td>Mary</td>
<td>s0412375</td>
</tr>
<tr>
<td>John</td>
<td>s0456782</td>
</tr>
</tbody>
</table>
Map Words to Length

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;this&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;is&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;the&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;time&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;and&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Map People to their Matric Nos: Wrong!

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>s0189034</td>
</tr>
<tr>
<td>Michael</td>
<td>s0289125</td>
</tr>
<tr>
<td>Peter</td>
<td>s0378435</td>
</tr>
<tr>
<td>Mary</td>
<td>s0412375</td>
</tr>
<tr>
<td>John</td>
<td>s0456782</td>
</tr>
</tbody>
</table>

NB  Keys must be unique.
A given key can only be mapped to **one value**.

However, type of value can be array, or some other object.
HashMap

Import HashMap

import java.util.HashMap;
HashMap

Import HashMap

```
import java.util.HashMap;
```

Declare HashMap

```
HashMap<String, Integer> map
    = new HashMap<String, Integer>();
```

- HashMap takes two type parameters.
- Here, String is type of key, Integer is type of value.
Goal: Given a string of words, derive an associative array that maps each word to its length.

1. Split the string on whitespace, to yield words.
2. For each word $w$, add it as a key, and associate it with value $w.length()$.
3. When we add the same key again, we overwrite the previous association — wasteful but harmless in this case.

**split() method of String**

```java
String sent = "this is the time and this is the record of the time";
String[] words = sent.split(" "); // split on whitespace
```
HashMap: Add and retrieve mappings

- put(Key, Value): put Value as the value of Key in wordLengths.

```java
HashMap<String, Integer> wordLengths = new HashMap<String, Integer>();
for (String word : words) {
    wordLengths.put(word, word.length());
}
```

- get(Key): get the value of Key in wordLengths.

```java
int wl = wordLengths.get("record"); // value is 6
```
HashMap: Add and retrieve mappings

wordLengths.keySet(): the set of keys in wordLengths.

[of, record, time, is, the, this, and]
HashMap: Add and retrieve mappings

`wordLengths.keySet()`: the set of keys in `wordLengths`.

[of, record, time, is, the, this, and]

**Q** How do we list all key-value pairs in a map?  
**A** Loop over the set of keys.

```java
for (String key : wordLengths.keySet()) {
    System.out.printf("%s => %s\n", key, wordLengths.get(key));
}
```

**Output**

```
of => 2
record => 6
time => 4
is => 2
the => 3
this => 4
```
HashMap: Printing

Output

System.out.println(wordLengths);

Output

{of=2, record=6, time=4, is=2, the=3, this=4, and=3}

Format is \{Key1=Value1, Key2=Value2, ... \}
ArrayList & HashMap

- Use ArrayList when you want your arrays to be able to grow, or you want to easily insert and remove items in the middle of an array.
- Use HashMap when you want to use keys other than a predetermined list of integers.
- For more on ArrayList and HashMap, look at the Java API: http://docs.oracle.com/javase/8/docs/api/
Reading

Java Tutorial
pp219-226, i.e. Chapter 7 Generics, stopping at Generic Methods.
pp423-505, i.e. Chapter 12 Collections, stopping at Algorithms.

In both cases, the book sections contain more material than we talked about. Remember the exam is open book! I don’t expect you to remember all the different kinds of collections, but I do expect you to be able to look them up and use them.